16-662 Robot Autonomy
Spring 2020
The Robotics Institute

Homework 1

Due: Friday January 31st, 11:59 PM

0 Prerequisites

0.1 Coding

All coding problems in this homework (and future homeworks) will be done in Python 3 (we use 3.6 in the solution code).

We include a requirements.txt with each homework listing the external packages you'd need to install to complete the homework. You may not use any package outside of those found in requirements.txt. We recommend using either virtualenv or conda to create a specific Python environment you'd use for this class.

Once you're in a Python 3 environment, installation of these packages can be done with the following command:

pip install -r requirements.txt

0.2 Submission

1. Please submit a zipfile [andrew ID].zip to Canvas that contains your code. Make sure the files run as expected and generate the plots that are asked for.
2. Please submit a PDF report to Gradescope with plots and answers to questions given below. Please use LaTeX for the report (we have included a template tex file).

1 Reinforcement Learning via Finite Difference (30 pts)

In this problem, you will explore using finite difference to train a policy for the Cart-Pole environment in OpenAI Gym:

All the code you need to complete this section are in the folder p1.

1.1 Getting familiar with CartPole Environment

This question uses run_cartpole.py

The CartPole environment is a simple testbed for reinforcement learning algorithms. The task has a 4-dimensional continuous observation space - each observation is a vector of the location of the cart, the speed of the cart, the angle of the pole, and the rotational speed of the pole. The task has a 2-item discrete action space - the policy can either choose to apply a leftward force by returning 0, or a rightward force by return 1.

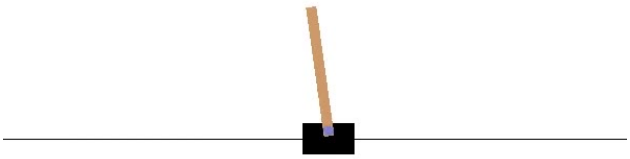

Figure 1: OpenAI Gym CartPole

The environment has a time horizon of 500, and it gives a reward of 1 every time step that the CartPole stays upright. The environment terminates when the CartPole falls below a certain threshold. So, the max reward a policy can achieve is 500.

Take a look at the run_policy function in run_cartpole.py to see how one interfaces with OpenAI Gym's env object. Specifically, note that we only have a black-box access to the environment via the env.step function.

In this problem, we will be using a simple linear policy of the following form:

$$\pi_\theta(s) = \mathbb{1}[\theta^\top s < 0] \tag{1}$$

where s is the 4-dimensional observation vector and θ is the 4-dimension parameter vector that parameterizes the policy.

1.1.1 Todo: Implement the Linear Policy (2 pts)

Implement the __call__ function in CartPoleLinearPolicy so that it performs Eq. 1

1.1.2 Todo: Run CartPole with Linear Policy (2 pts)

Fill in some non-zero numbers for the params variable and observe cartpole behavior and reward.

For example, try making the CartPole always go left or right. You might even be able to manually find a set of parameters that achieves max reward!

Include in your report what parameters you tried and the behavior they obtained.

1.1.3 Todo: Sample Random Policies and Observe Behavior. (1 pt)

Sample 1000 random policies, each with parameters whose entries are drawn from a uniform distribution between -1 and 1.

In your report, include:

1. Histogram of all the rewards obtained by the 1000 random policies.

2. Percentage of the random policies achieved max reward.

If your policy class is implemented correctly, run_cartpole.py should do this sampling and generate the histogram automatically.

1.2 Finite Difference

This question uses finite_diff.py

While you might've stumbled across a policy that achieves max reward either by manual tuning or random sampling, this is very inefficient, especially for problems with larger observation and action spaces. A better way to find this policy is by starting at some initial guess for the policy parameters and then optimizing the parameters to maximize the reward.

We will do this policy search via gradient ascent in the next section - more details will follow. For now, we will implement a finite difference utility function that approximates the gradient of a given function by perturbing the inputs. This will become useful when we need to compute the gradient of the reward with respect to the policy parameters.

1.2.1 Todo: Implement scalar_finite_diff function (10 pts)

This function takes in a three parameters - f, x, and h. It returns the approximated gradient of f at x, and h has the same dimension as x and contains the magnitude of the deltas used to perturb x and compute the finite difference.

For now we restrict f to be a scalar function, so the gradient should have the same dimension as x. If $x \in \mathbb{R}^N$, then:

$$\nabla_x f = [\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_N}] \tag{2}$$

There are many ways to implement finite difference. We will go with a simple way that works for this problem, the central method:

$$\left.\frac{\partial f(x)}{\partial x_i}\right|_{x=x'} \approx \frac{f(x + h_i v^{(i)}) - f(x - h_i v^{(i)})}{2h_i} \tag{3}$$

where $v^{(i)}$ has the same dimension as x and $v_j^{(i)} = \mathbb{1}[i == j]$.

Implement the scalar_finite_diff function using the central method, and run finite_diff.py - this will run the unit test for your finite difference function. Make sure the unit tests pass!

1.3 Training CartPole Policy

This question uses train_cartpole.py

Now we're ready to train a policy for CartPole! Let $g(\theta) = \mathbb{E}\, R(\pi_\theta)$ be a function that takes in the parameters for a policy and outputs its expected reward over the environment. The expectation is present because, even though the CartPole environment is deterministic, the initial conditions of the CartPole system is drawn from a distribution, so a particular parameter might get lucky or unlucky over one single rollout given a particular initial condition.

Once we have $g(\theta)$, we can perform the following gradient update to search for a θ that achieves higher rewards:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} + \alpha \nabla_\theta g(\theta^{(t)}) \tag{4}$$

where t denotes the training iteration number and α is the learning rate.

1.3.1 Todo: Implement Parameter Evaluation Function (5 pts)

Implement the eval_params function which approximates $g(\theta)$ by running the policy some fixed number of times.

1.3.2 Todo: Implement Gradient Ascent Parameter Updates (5 pts)

Implement the gradient ascent update step and record the policy performance after each update.

1.3.3 Todo: Tune Learning Rate and Plot Learning Curve (5 pts)

Like many things in learning, this policy search procedure can be quite sensitive to the learning rate parameter. Once you've implemented the parameter evaluation function and the gradient updates, now it's time to tune the learning rate so that the training procedure succeeds.

Please find a learning rate that allows the training procedure to converge with 7 or less iterations. Convergence means the policy reaches the max reward of 500 and never goes below it in the iterations that follow.

Include in your report:

1. The learning rate you used.
2. The final policy parameters found by your gradient ascent.
3. A training curve plot, where the x-axis is the training iterations and the y-axis is the reward of the policy achieved at that iteration.

2 Supervised Learning: Classification via Logistic Regression (30 pts)

This question uses logistic_regression.py in the folder p2.

In this question, you will implement multi-class logistic regression to classify the type of food a robot is cutting via sound and force data.

There are 7 classes in total with the following class labels:

0. Apple
1. Carrot
2. Cucumber
3. Frozen Lemon

4. Celery
5. Onion
6. Potato

During data collection, a Franka robot holding a knife performs a cutting motion on a fruit or vegetable on a cutting board:

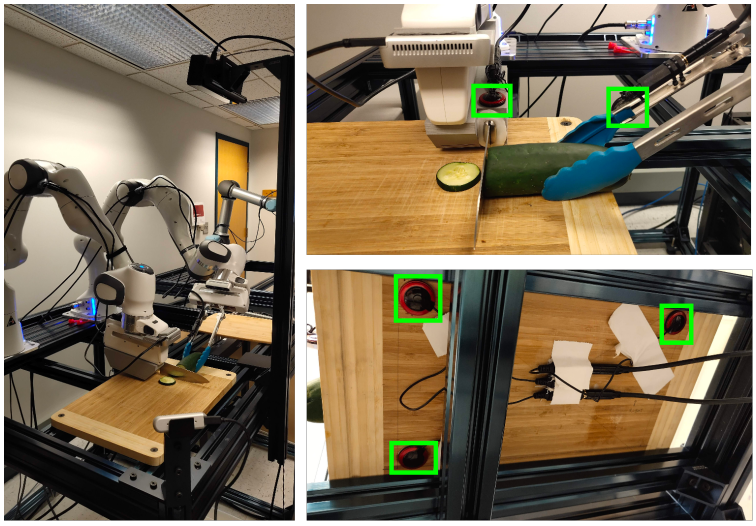

Figure 2: Food Cutting Data Collection with Franka Robot

The data includes 10658 samples. You can listen to the example audio recordings of each type of food in the audio folder. We will not be using these audio files directly. Instead, we've provided data.npz, which contains features extracted from the raw audio recordings.

Each data sample consists of features extracted from 4 contact microphones (3 on the cutting board, 1 on the end-effector), as well as the end-effector force and torque readings. Each microphone has 40 features (mel-frequency cepstral coefficients) corresponding to 40 0.1s timesteps, and there are an additional $6 \times 10 = 60$ features corresponding to 10 timesteps of end-effector force and torque readings in 6D. Together, each data sample consists of 220 features.

We've provided the skeleton code to load data, run the classifier, and produce the training curve plot.

2.1 Two-Class Logistic Regression

Logistic Regression is typically used for two-class classification, but we can extend it to multiple classes easily by training multiple classifiers, one for each class. More details on this in the next section.

In Logistic Regression, we have a model that predicts the probability of the sample being a positive class (1 as opposed to 0):

$$P(y|x) = \frac{1}{1 + e^{w^\top x + b}} = \sigma(w^\top x + b) \tag{5}$$

where y is the binary label, x is the feature vector of a data sample, and w and b are the parameters of this model. σ is called the sigmoid function. Note the addition of b, the bias term, is needed so that we can handle the case where the decision boundary lies outside of the origin in the feature space. For ease of writing and coding, we will augment x with a constant feature of 1, so the exponentiated term can be written as:

$$w^\top x + b \to [w, b]^\top [x, 1] \to \tilde{w}^\top \tilde{x} \tag{6}$$

Once we have the model, the next thing we need is the objective function over which to optimize the parameters of the model. The objective we use will be log likelihood. Denote x_i and y_i to be the ith feature-label pair in a dataset of N samples. Then, the log likelihood l of the model parameters over the dataset is:

$$l(\tilde{w}) = \sum_{i=1}^{N} y_i \log \sigma(\tilde{w}^\top \tilde{x_i}) + (1 - y_i) \log(1 - \sigma(\tilde{w}^\top \tilde{x_i})) \tag{7}$$

We will optimize $l(\tilde{w})$ w.r.t $\tilde{w}$ via gradient ascent.

2.1.1 Todo: Derive the Logistic Regression Gradient Update Formula (10 pts)

Please derive the expression for $\nabla_{\tilde{w}} l(\tilde{w})$.

2.2 Implement TwoClassLogisticRegressionClassifier

2.2.1 Todo: Implement the sigmoid function in _sigmoid (2 pts)

This applies the sigmoid function over an array of numbers.

2.2.2 Todo: Implement the predict_prob function (3 pts)

Given an input matrix $X \in \mathbb{R}^{N \times M}$, where N is the number of samples and M the dimension of the features, this returns the predicted probability of positive class for each of the N samples. Note you need to augment X with the constant 1 feature for the bias term to work.

2.2.3 Todo: Implement the gradient ascent update step (5 pts)

Use the expression you derived in the previous part to implement the gradient ascent update:

$$\tilde{w}^{(t+1)} \leftarrow \tilde{w}^{(t)} + \alpha \nabla_{\tilde{w}} l(\tilde{w}^{(t)}) \tag{8}$$

2.3 Multi-Class Logistic Regression

To use Logistic Regression for C classes, we simply train C two-class Logistic Regression classifiers. The cth classifier will be trained with identifying positive label for the cth class, and negative label for all the other classes. In practice, this means generating C sets of two-class training labels specific for each classifier.

2.3.1 Todo: Implement the fit function in MultiClassLogisticRegressionClassifier (5 pts)

In the inner loop of the fit function, create a set of new two-class labels for the cth classifier, and fit the cth classifier for 1 step.

2.3.2 Todo: Run Training Script (5 pts)

If everything's coded correctly, you should be able to run logistic_regression.py without errors, and it should perform training and also output training curve plots.

You should see that the training accuracy reaches above 99% in 70 iterations or less.

Include in your report:

1. Plot of the training curve.
2. Final train and test accuracy.